Iozone Filesystem Benchmark

IOzone is a filesystem benchmark tool. The benchmark generates and measures a variety of file operations. Iozone has been ported to many machines and runs under many operating systems. This document will cover the many different types of operations that are tested as well as coverage of all of the command line options.
Iozone is useful for determining a broad filesystem analysis of a vendor's computer platform. The benchmark tests file I/O performance for the following operations.

> *Read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read/write, pread/pwrite variants, aio_read, aio_write, mmap,*

While computers are typically purchased with an application in mind it is also likely that over time the application mix will change. Many vendors have enhanced their operating systems to perform well for some frequently used applications. Although this accelerates the I/O for those few applications it is also likely that the system may not perform well for other applications that were not targeted by the operating system. An example of this type of enhancement is: Database. Many operating systems have tested and tuned the filesystem so it works well with databases. While the database users are happy, the other users may not be so happy as the entire system may be giving all of the system resources to the database users at the expense of all other users. As time rolls on the system administrator may decide that a few more office automation tasks could be shifted to this machine. The load may now shift from a random reader application (database) to a sequential reader. The users may discover that the machine is very slow when running this new application and become dissatisfied with the decision to purchase this platform. By using Iozone to get a broad filesystem performance coverage the buyer is much more likely to see any hot or cold spots and pick a platform and operating system that is more well balanced.

Features:

- ANSII 'C' source.
- POSIX async I/O.
- Mmap() file I/O.
- Normal file I/O.
- Single stream measurement.
- Multiple stream measurement.
- POSIX pthreads.
- Multi-process measurement.
- Excel importable output for graph generation.
- I/O Latency data for plots.
- 64-bit compatible source.
- Large file compatible.
- Stonewalling in throughput tests to eliminate straggler effects.
- Processor cache size configurable.
- Selectable measurements with fsync, O_SYNC.
- Options targeted for testing over NFS.

Building IOzone

Once you have obtained the source for IOzone you should have 12 files.

- iozone.c (source code)
- libasync.c (source code)
- makefile (makefile)
- libbif.c (source code)
- Iozone_msword_98.doc (documentation in Word format)
- iozone.1 (documentation in nroff format)

- gnuplot.dem (sample gnuplot file)
- gnuplotps.dem (sample gnuplot file that generates postscript output)
- read_telemetry (sample file for read telemetry file)
- write_telemetry (sample file for write telemetry file)
- Run_rules.doc (run rules to get reasonable results)
- Changes.txt (log of changes to Iozone since its beginning)

Type: make
The makefile will display a list of supported platforms. Pick the one that matches your configuration and then type: make target
That's it. You're done. There is no need to have any install procedures as IOzone creates all of its files in the current working directory. Just copy Iozone to wherever you wish to test the filesystem performance and then run it. Or you can use the **–f** command line option to specify a target path, for example, a path/filename in a new filesystem.

Before you run Iozone please read the run rules at the bottom of this document.

Examples of running Iozone:

The simplest way to get started is to try the automatic mode.

```
Iozone –a
```

If you wish to generate graphs then you may wish to turn on Excel mode.

```
Iozone –Ra                    ( Output can be imported using space and tab delimited)
    Or
Iozone –Rab output.wks  ( Output file "output.wks" is a binary format spreadsheet)
```

If you have more than 512 Mbytes of memory then you need to increase the maximum file size to a larger value. For example if your system has 1 Gbyte of memory then you would want to try something like:

```
Iozone –Ra –g 2G
```

If you only care about read/write and do not wish to spend the time to perform all of the tests, then you may wish to limit the testing like:

```
Iozone –Ra –g 2G –i 0 –i 1
```

If you are running Iozone over NFS on an NFS client then you may wish to use:

```
Iozone –Rac
```

This tells Iozone to include the close() in the measurement. This may be needed if the client is running NFS version 3. Including the close() helps to reduce the client side cache effects of NFS version 3. If you use a file size that is larger than the amount of memory in the client then the 'c' flag is not needed.

Definitions of the tests

Write: This test measures the performance of writing a new file. When a new file is written not only does the data need to be stored but also the overhead information for keeping track of where the data is located on the storage media. This overhead is called the “metadata” It consists of the directory information, the space allocation and any other data associated with a file that is not part of the data contained in the file. It is normal for the initial write performance to be lower than the performance of re-writing a file due to this overhead information.

Re-write: This test measures the performance of writing a file that already exists. When a file is written that already exists the work required is less as the metadata already exists. It is normal for the rewrite performance to be higher than the performance of writing a new file.

Read: This test measures the performance of reading an existing file.

Re-Read: This test measures the performance of reading a file that was recently read. It is normal for the performance to be higher as the operating system generally maintains a cache of the data for files that were recently read. This cache can be used to satisfy reads and improves the performance.

Random Read: This test measures the performance of reading a file with accesses being made to random locations within the file. The performance of a system under this type of activity can be impacted by several factors such as: Size of operating system’s cache, number of disks, seek latencies, and others.

Random Write: This test measures the performance of writing a file with accesses being made to random locations within the file. Again the performance of a system under this type of activity can be impacted by several factors such as: Size of operating system’s cache, number of disks, seek latencies, and others.

Random Mix: This test measures the performance of reading and writing a file with accesses being made to random locations within the file. Again the performance of a system under this type of activity can be impacted by several factors such as: Size of operating system’s cache, number of disks, seek latencies, and others. This test is only available in throughput mode. Each thread/process runs either the read or the write test. The distribution of read/write is done on a round robin basis. More than one thread/process is required for proper operation.

Backwards Read: This test measures the performance of reading a file backwards. This may seem like a strange way to read a file but in fact there are applications that do this. MSC Nastran is an example of an application that reads its files backwards. With MSC Nastran, these files are very large (Gbytes to Tbytes in size). Although many operating systems have special features that enable them to read a file forward more rapidly, there are very few operating systems that detect and enhance the performance of reading a file backwards.

Record Rewrite: This test measures the performance of writing and re-writing a particular spot within a file. This hot spot can have very interesting behaviors. If the size of the spot is small enough to fit in the CPU data cache then the performance is very high. If the size of the spot is bigger than the CPU data cache but still fits in the TLB then one gets a different level of performance. If the size of the spot is larger than the CPU data cache and larger than the TLB but still fits in the operating system cache then one gets another level of performance, and if the size of the spot is bigger than the operating system cache then one gets yet another level of performance.

Strided Read: This test measures the performance of reading a file with a strided access behavior. An example would be: Read at offset zero for a length of 4 Kbytes, then seek 200 Kbytes, and then read for a length of 4 Kbytes, then seek 200 Kbytes and so on. Here the pattern is to read 4 Kbytes and then

Seek 200 Kbytes and repeat the pattern. This again is a typical application behavior for applications that have data structures contained within a file and is accessing a particular region of the data structure. Most operating systems do not detect this behavior or implement any techniques to enhance the performance under this type of access behavior.
This access behavior can also sometimes produce interesting performance anomalies. An example would be if the application's stride causes a particular disk, in a striped file system, to become the bottleneck.

Fwrite: This test measures the performance of writing a file using the library function fwrite(). This is a library routine that performs buffered write operations. The buffer is within the user's address space. If an application were to write in very small size transfers then the buffered & blocked I/O functionality of fwrite() can enhance the performance of the application by reducing the number of actual operating system calls and increasing the size of the transfers when operating system calls are made. This test is writing a new file so again the overhead of the metadata is included in the measurement.

Frewrite: This test measures the performance of writing a file using the library function fwrite(). This is a library routine that performs buffered & blocked write operations. The buffer is within the user's address space. If an application were to write in very small size transfers then the buffered & blocked I/O functionality of fwrite() can enhance the performance of the application by reducing the number of actual operating system calls and increasing the size of the transfers when operating system calls are made. This test is writing to an existing file so the performance should be higher as there are no metadata operations required.

Fread: This test measures the performance of reading a file using the library function fread(). This is a library routine that performs buffered & blocked read operations. The buffer is within the user's address space. If an application were to read in very small size transfers then the buffered & blocked I/O functionality of fread() can enhance the performance of the application by reducing the number of actual operating system calls and increasing the size of the transfers when operating system calls are made.

Freread: This test is the same as fread above except that in this test the file that is being read was read in the recent past. This should result in higher performance as the operating system is likely to have the file data in cache.

Specialized tests:

Mmap: Many operating systems support the use of mmap() to map a file into a user's address space. Once this mapping is in place then stores to this location in memory will result in the data being stored going to a file. This is handy if an application wishes to treat files as chunks of memory. An example would be to have an array in memory that is also being maintained as a file in the files system.
The semantics of mmap files is somewhat different than normal files. If a store to the memory location is done then no actual file I/O may occur immediately. The use of the msyc() with the flags MS_SYNC, and MS_ASYNC control the coherency of the memory and the file. A call to msync() with MS_SYNC will force the contents of memory to the file and wait for it to be on storage before returning to the application. A call to msync() with the flag MS_ASYNC tells the operating system to flush the memory out to storage using an asynchronous mechanism so that the application may return into execution without waiting for the data to be written to storage.
This test measures the performance of using the mmap() mechanism for performing I/O.

Async I/O: Another mechanism that is supported by many operating systems for performing I/O is POSIX async I/O. The application uses the POSIX standard async I/O interfaces to accomplish this. Example: aio_write(), aio_read(), aio_error(). This test measures the performance of the POSIX async I/O mechanism.

Command Line options:

The following is the output from the built in help. Each option's purpose is explained in this section of the manual.

```
Usage: iozone [-s filesize_Kb] [-r record_size_Kb ] [-f [path]filename]
        [-i test] [-E] [-p] [-a] [-A] [-z] [-Z] [-m] [-M] [-t children] [-h] [-o]
        [-l min_number_procs] [-u max_number_procs] [-v] [-R] [-x]
        [-d microseconds] [-F path1 path2...] [-V pattern] [-j stride]
        [-T] [-C] [-B] [-D] [-G] [-I] [-H depth] [-k depth] [-U mount_point]
        [-S cache_size] [-O] [-K] [-L line_size] [-g max_filesize_Kb]
        [-n min_filesize_Kb] [-N] [-Q] [-P start_cpu] [-c] [-e] [-b filename]
        [-J milliseconds] [-X filename] [-Y filename] [-w] [-W]
        [-y min_recordsize_Kb] [-q max_recordsize_Kb] [-+m filename]
        [-+u ] [ -+d ] [-+p percent_read] [-+r] [-+t ] [-+A #]
```

What do they all mean ?

-a
Used to select full automatic mode. Produces output that covers all tested file operations for record sizes of 4k to 16M for file sizes of 64k to 512M.

-A
This version of automatic mode provides more coverage but consumes a bunch of time. The **–a** option will automatically stop using transfer sizes less than 64k once the file size is 32 MB or larger. This saves time. The **–A** option tells Iozone that you are willing to wait and want dense coverage for small transfers even when the file size is very large. **NOTE:** This option is deprecated in Iozone version 3.61. Use **–az –i 0 –i 1** instead.

-b filename
Iozone will create a binary file format file in Excel compatible output of results.

-B
Use mmap() files. This causes all of the temporary files being measured to be created and accessed with the mmap() interface. Some applications prefer to treat files as arrays of memory. These applications mmap() the file and then just access the array with loads and stores to perform file I/O.

-c
Include close() in the timing calculations. This is useful only if you suspect that close() is broken in the operating system currently under test. It can be useful for NFS Version 3 testing as well to help identify if the nfs3_commit is working well.

-C
Show bytes transferred by each child in throughput testing. Useful if your operating system has any starvation problems in file I/O or in process management.

-d #
Microsecond delay out of barrier. During the throughput tests all threads or processes are forced to a barrier before beginning the test. Normally, all of the threads or processes are released at the same moment. This option allows one to delay a specified time in microseconds between releasing each of the processes or threads.

-D
Use msync(MS_ASYNC) on mmap files. This tells the operating system that all the data in

the mmap space needs to be written to disk asynchronously.

-e
Include flush (fsync,fflush) in the timing calculations

-E
Used to select the extension tests. Only available on some platforms. Uses pread interfaces.

-f filename
Used to specify the filename for the temporary file under test. This is useful when the unmount option is used. When testing with unmount between tests it is necessary for the temporary file under test to be in a directory that can be unmounted. It is not possible to unmount the current working directory as the process Iozone is running in this directory.

-F filename filename filename …
Specify each of the temporary file names to be used in the throughput testing. The number of names should be equal to the number of processes or threads that are specified.

-g #
Set maximum file size (in Kbytes) for auto mode.

-G
Use msync(MS_SYNC) on mmap files. This tells the operating system that all the data in the mmap space needs to be written to disk synchronously.

-h
Displays help screen.

-H #
Use POSIX async I/O with # async operations. Iozone will use POSIX async I/O with a bcopy from the async buffers back into the applications buffer. Some versions of MSC NASTRAN perform I/O this way. This technique is used by applications so that the async I/O may be performed in a library and requires no changes to the applications internal model.

-i #
Used to specify which tests to run. (0=write/rewrite, 1=read/re-read, 2=random-read/write 3=Read-backwards, 4=Re-write-record, 5=stride-read, 6=fwrite/re-fwrite, 7=fread/Re-fread, 8=random mix, 9=pwrite/Re-pwrite, 10=pread/Re-pread, 11=pwritev/Re-pwritev, 12=preadv/Re-preadv).
One will always need to specify 0 so that any of the following tests will have a file to measure.
-i # -i # -i # is also supported so that one may select more than one test.

-I
Use VxFS VX_DIRECT for all file operations. Tells the VXFS filesystem that all operations to the file are to bypass the buffer cache and go directly to disk.

-j #
Set stride of file accesses to (# * record size). The stride read test will read records at this stride.

-J # (in milliseconds)
Perform a compute delay of this many milliseconds before each I/O operation. See also **-X** and **-Y** for other options to control compute delay.

-k #

Use POSIX async I/O (no bcopy) with # async operations. Iozone will use POSIX async I/O and will not perform any extra bcopys. The buffers used by Iozone will be handed to the async I/O system call directly.

-K

Generate some random accesses during the normal testing.

-l #

Set the lower limit on number of processes to run. When running throughput tests this option allows the user to specify the least number of processes or threads to start. This option should be used in conjunction with the **-u** option.

-L #

Set processor cache line size to value (in bytes). Tells Iozone the processor cache line size. This is used internally to help speed up the test.

-m

Tells Iozone to use multiple buffers internally. Some applications read into a single buffer over and over. Others have an array of buffers. This option allows both types of applications to be simulated. Iozone’s default behavior is to re-use internal buffers. This option allows one to override the default and to use multiple internal buffers.

-M

Iozone will call uname() and will put the string in the output file.

-n #

Set minimum file size (in Kbytes) for auto mode.

-N

Report results in microseconds per operation.

-o

Writes are synchronously written to disk. (O_SYNC). Iozone will open the files with the O_SYNC flag. This forces all writes to the file to go completely to disk before returning to the benchmark.

-O

Give results in operations per second.

-p

This purges the processor cache before each file operation. Iozone will allocate another internal buffer that is aligned to the same processor cache boundary and is of a size that matches the processor cache. It will zero fill this alternate buffer before beginning each test. This will purge the processor cache and allow one to see the memory subsystem without the acceleration due to the processor cache.

-P #

Bind processes/threads to processors, starting with this cpu #. Only available on some platforms. The first sub process or thread will begin on the specified processor. Future processes or threads will be placed on the next processor. Once the total number of cpus is exceeded then future processes or threads will be placed in a round robin fashion.

-q #

Set maximum record size (in Kbytes) for auto mode. One may also specify
-q #k (size in Kbytes) or **-q #m** (size in Mbytes) or **-q #g** (size in Gbytes).
See **–y** for setting minimum record size.

-Q

Create offset/latency files. Iozone will create latency versus offset data files that can be imported with a graphics package and plotted. This is useful for finding if certain offsets have very high latencies. Such as the point where UFS will allocate its first indirect block. One can see from the data the impacts of the extent allocations for extent based filesystems with this option.

-r #

Used to specify the record size, in Kbytes, to test. One may also specify
-r #k (size in Kbytes) or **-r #m** (size in Mbytes) or **-r #g** (size in Gbytes).

-R

Generate Excel report. Iozone will generate an Excel compatible report to standard out. This file may be imported with Microsoft Excel (space delimited) and used to create a graph of the filesystem performance. Note: The 3D graphs are column oriented. You will need to select this when graphing as the default in Excel is row oriented data.

-s #

Used to specify the size, in Kbytes, of the file to test. One may also specify
-s #k (size in Kbytes) or **-s #m** (size in Mbytes) or **-s #g** (size in Gbytes).

-S #

Set processor cache size to value (in Kbytes). This tells Iozone the size of the processor cache. It is used internally for buffer alignment and for the purge functionality.

-t #

Run Iozone in a throughput mode. This option allows the user to specify how many threads or processes to have active during the measurement.

-T

Use POSIX pthreads for throughput tests. Available on platforms that have POSIX threads.

-u #

Set the upper limit on number of processes to run. When running throughput tests this option allows the user to specify the greatest number of processes or threads to start. This option should be used in conjunction with the **-l** option.

-U mountpoint

Mount point to unmount and remount between tests. Iozone will unmount and remount this mount point before beginning each test. This guarantees that the buffer cache does not contain any of the file under test.

-v

Display the version of Iozone.

-V #

Specify a pattern that is to be written to the temporary file and validated for accuracy in each of the read tests.

-w

Do not unlink temporary files when finished using them. Leave them present in the filesystem.

-W

Lock files when reading or writing.

-x

Turn off stone-walling. Stonewalling is a technique used internally to Iozone. It is used during the throughput tests. The code starts all threads or processes and then stops them on a barrier. Once they are all ready to start then they are all released at the same time. The moment that any of the threads or processes finish their work then the entire test is terminated and throughput is calculated on the total I/O that was completed up to this point. This ensures that the entire measurement was taken while all of the processes or threads were running in parallel. This flag allows one to turn off the stonewalling and see what happens.

-X filename

Use this file for write telemetry information. The file contains triplets of information: Byte offset, size of transfer, compute delay in milliseconds. This option is useful if one has taken a system call trace of the application that is of interest. This allows Iozone to replicate the I/O operations that this specific application generates and provide benchmark results for this file behavior. (if column 1 contains # then the line is a comment)

-y #

Set minimum record size (in Kbytes) for auto mode. One may also specify **-y #k** (size in Kbytes) or **-y #m** (size in Mbytes) or **-y #g** (size in Gbytes). See **–q** for setting maximum record size.

-Y filename

Use this file for read telemetry information. The file contains triplets of information: Byte offset, size of transfer, compute delay in milliseconds. This option is useful if one has taken a system call trace of the application that is of interest. This allows Iozone to replicate the I/O operations that this specific application generates and provide benchmark results for this file behavior. (if column 1 contains # then the line is a comment)

-z

Used in conjunction with **-a** to test all possible record sizes. Normally Iozone omits testing of small record sizes for very large files when used in full automatic mode. This option forces Iozone to include the small record sizes in the automatic tests also.

-Z

Enable mixing mmap I/O and file I/O.

-+m filename

Use this file to obtain the configuration information of the clients for cluster testing. The file contains one line for each client. Each line has three fields. The fields are space delimited. A # sign in column zero is a comment line. The first field is the name of the client. The second field is the path, on the client, for the working directory where Iozone will execute. The third field is the path, on the client, for the executable Iozone.

To use this option one must be able to execute commands on the clients without being challenged for a password. Iozone will start remote execution by using “rsh”.

-+u

Enable CPU utilization mode.

-+d

Enable diagnostic mode. In this mode every byte is validated. This is handy if one suspects a broken I/O subsystem.

-+p percent_read

Set the percentage of the thread/processes that will perform random read testing. Only valid in throughput mode and with more than 1 process/thread.

-+r

Enable O_RSYNC and O_SYNC for all I/O testing.

-+t

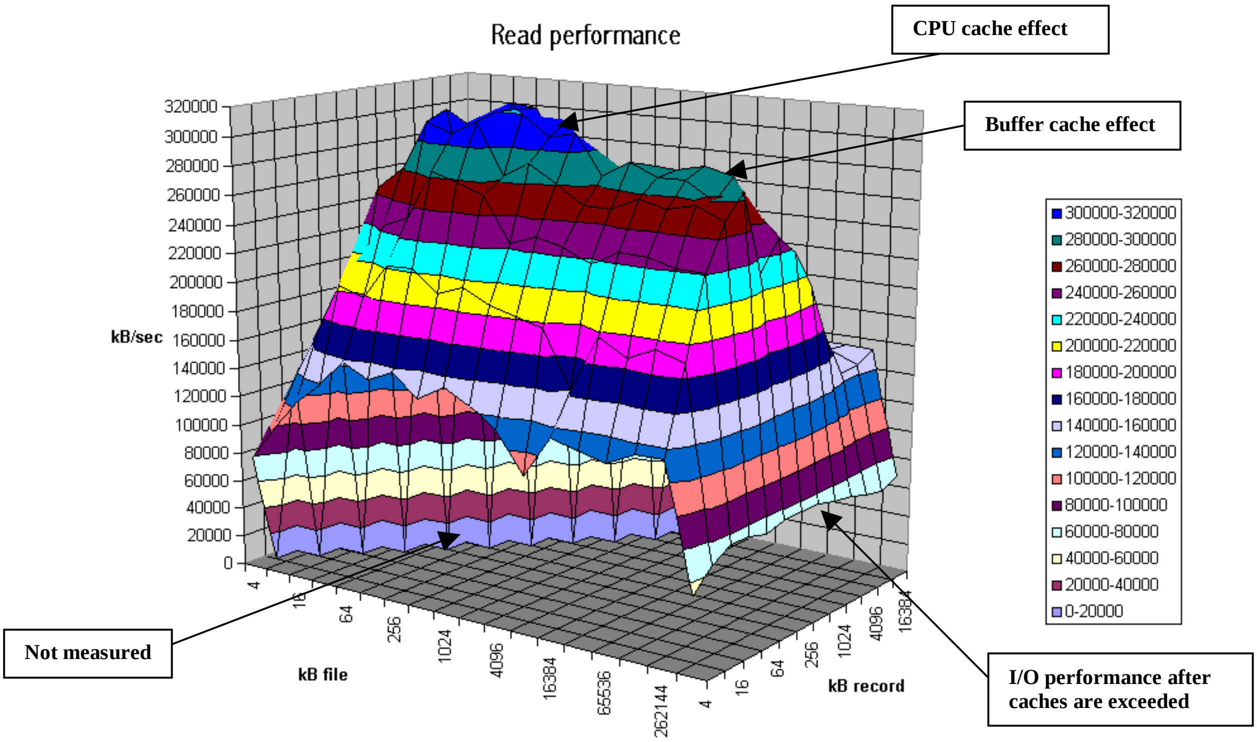
Enable network performance test. Requires -+m

-+A

Enable madvise. 0 = normal, 1=random, 2=sequential, 3=dontneed, 4=willneed.
For use with options that activate mmap() file I/O. See: -B

What can I see:

The following are some graphs that were generated from the Iozone output files.



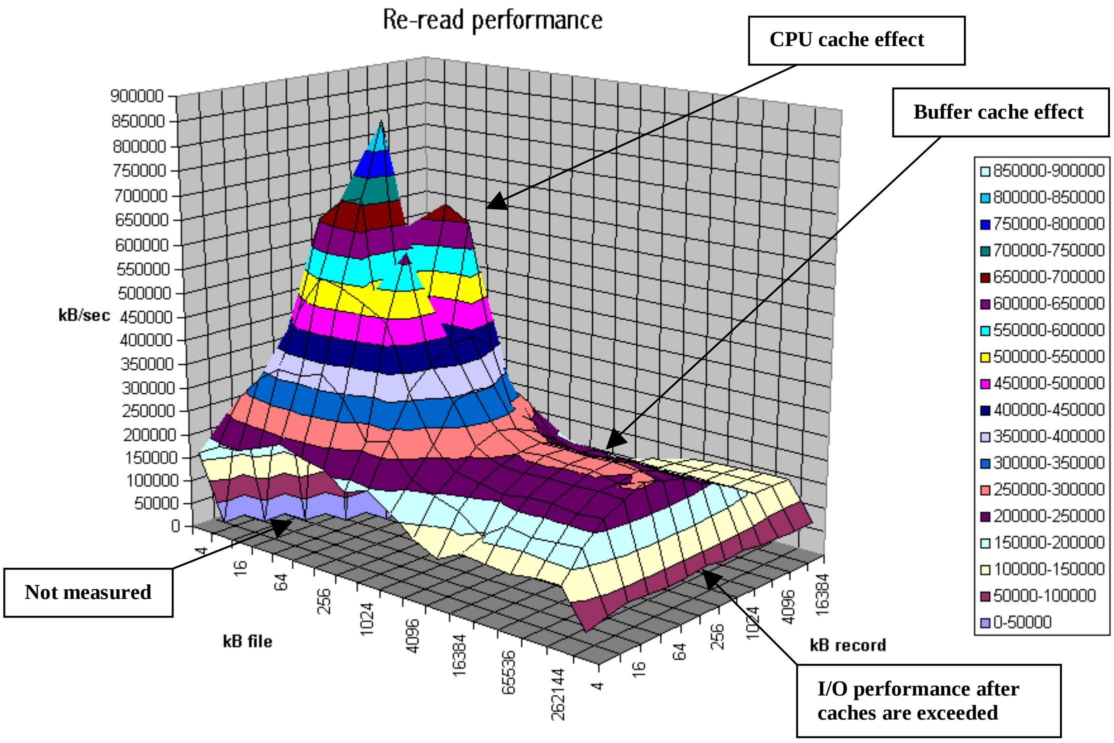
CPU cache effect

Buffer cache effect

Not measured

I/O performance after caches are exceeded

From the graph above one can clearly see the buffer cache helping out for file sizes that are less than 256MB but after that the actual disk I/O speed can be seen. Also note that the processor cache effects can be seen for file sizes of 16 Kbytes to 1Mbyte.



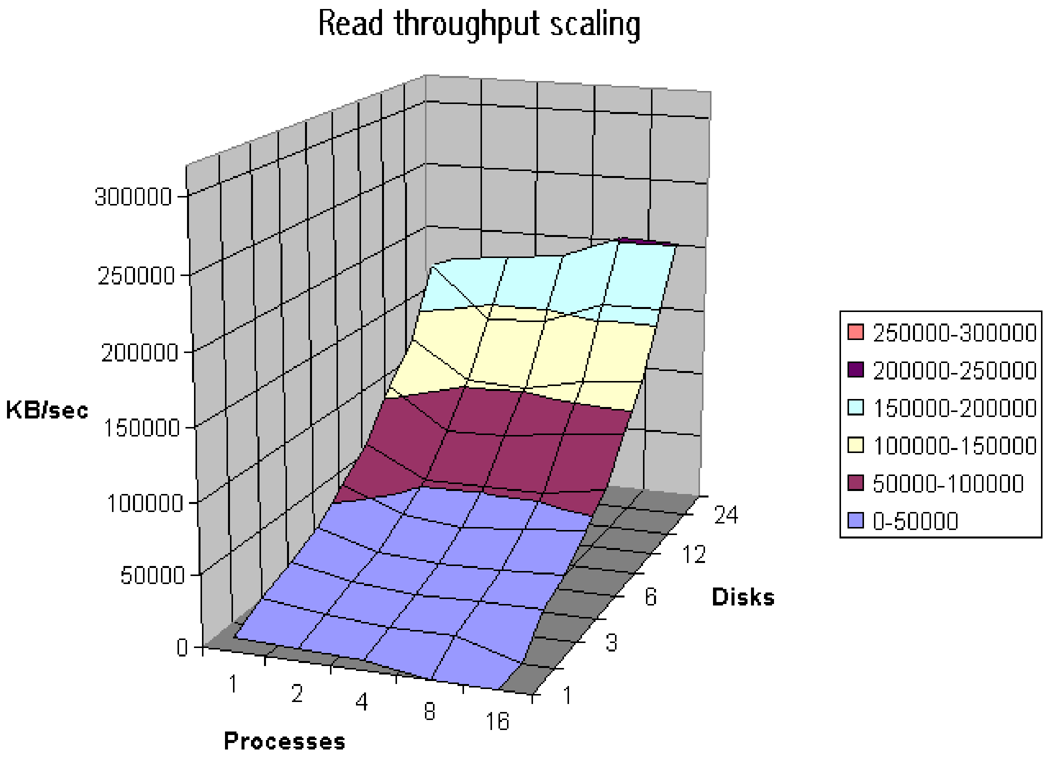
CPU cache effect

Buffer cache effect

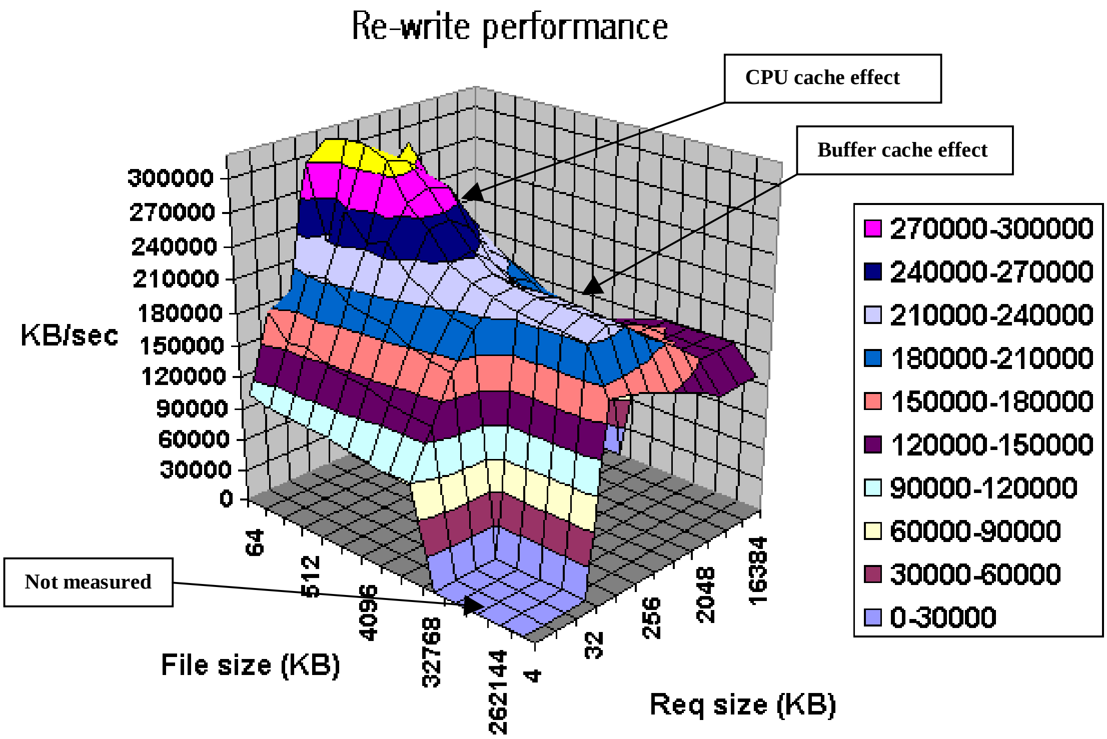
Not measured

I/O performance after caches are exceeded

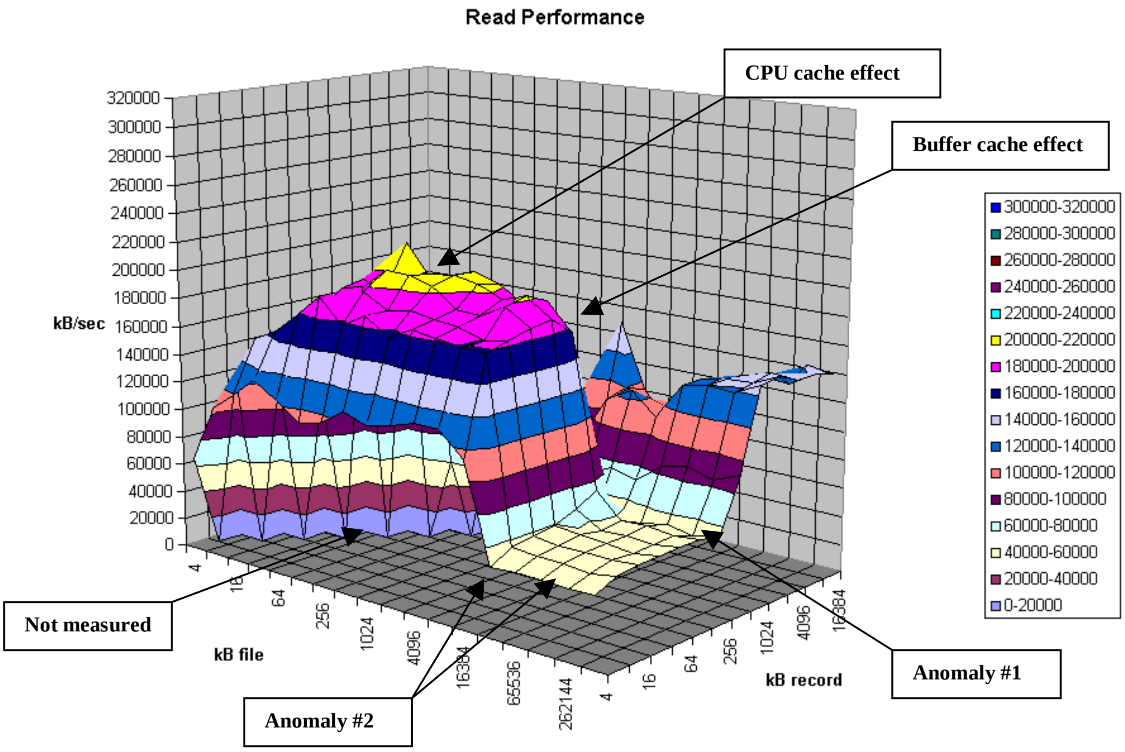
The graph above is displaying the impact of re-reading a file. Notice that the processor cache is now very important and causes the sharp peak. The next plateau to the right is buffer cache and finally above 256MB the file no longer fits in the buffer cache and real spindle speeds can be seen.



The graph above was created by running Iozone multiple times and then graphing the combination of the results. Here the graph is showing the throughput performance as a function of processes and number of disks participating in a filesystem. (disk striping) The good news is that on this system as one adds disks the throughput increases. Not all platforms scale so well.

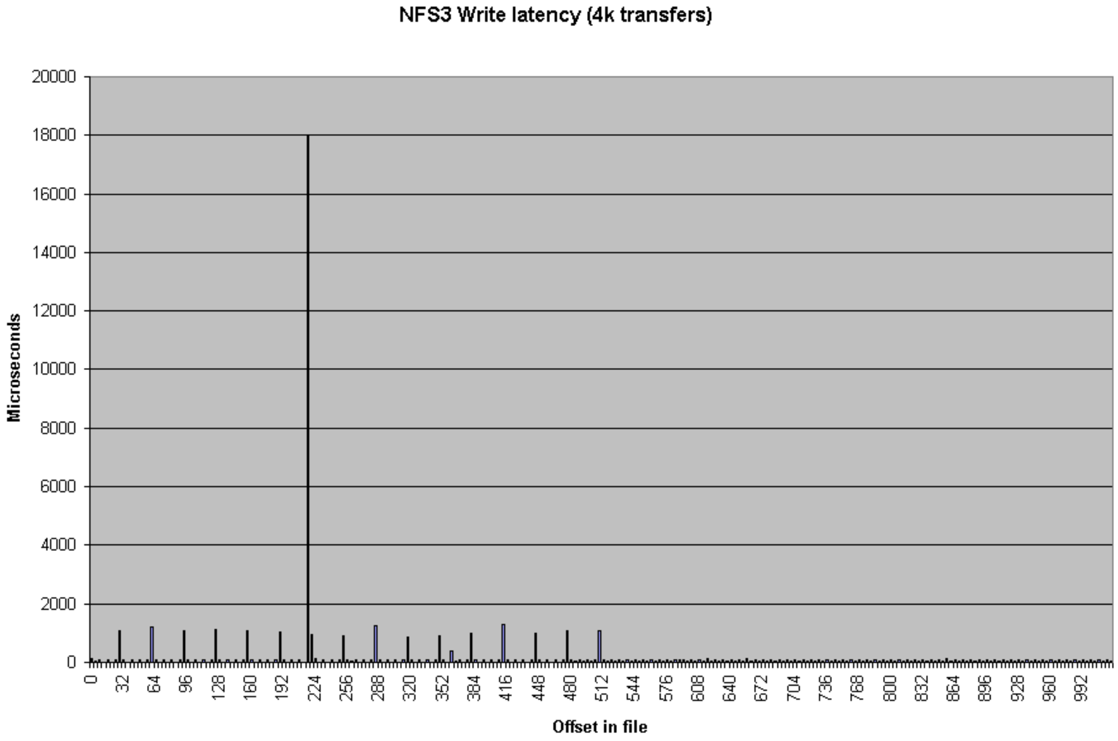


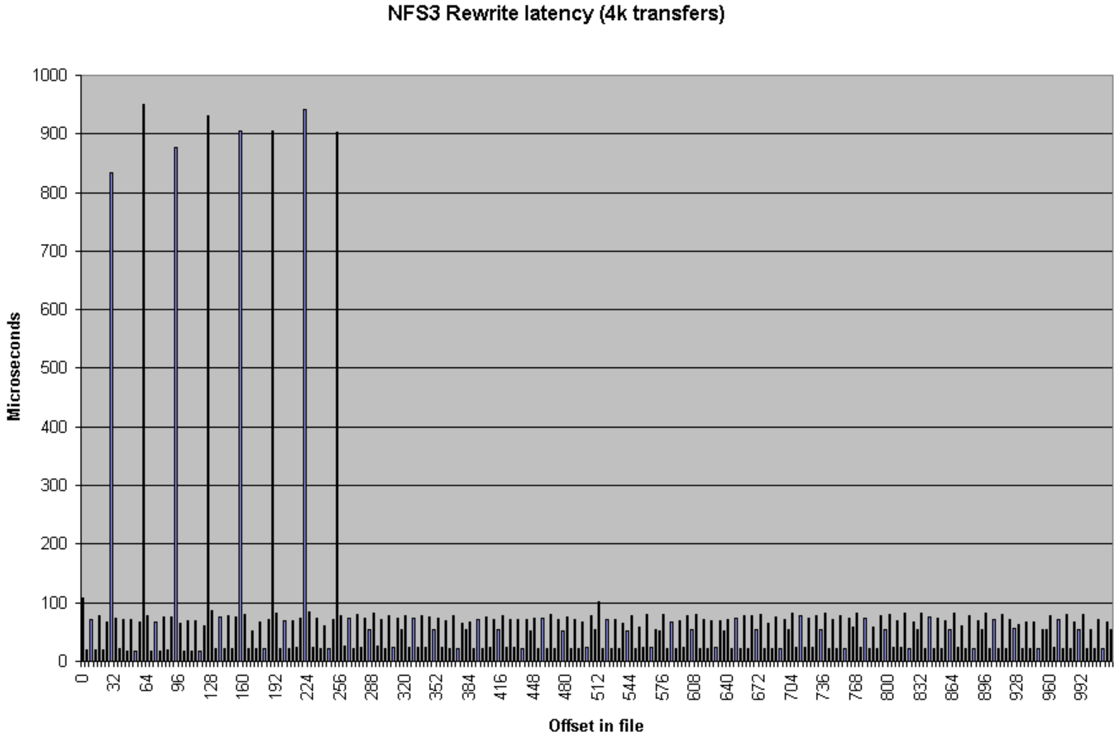
The graph above shows single stream performance where file size and request size are changed. The place on the lower right that touches the floor of the graph is not actual data. Excel graphs empty cells as containing a zero. This run was taken with the –a option. If one used the –A option then the area that was not tested would have been tested and had real values. Normally this is not a desirable area to test because it is very time consuming to write a 512MB file in 4k transfer sizes. The –a option in Iozone tells Iozone to discontinue use of transfer sizes less than 64k once the file size is 32MB or bigger. This saves quite a bit of time. Notice the ridge that runs from the top left to the lower right down the center of the graph. This is where the request size fits in the processor cache. For file sizes less than the size of the processor cache you can see the rise in performance as well. When both the file size and the transfer size is less than the processor cache it rises even higher. Although interesting to see, it is unlikely that you will be able to get applications to never write files that are bigger than the processor cache ☺ However it might be possible to get applications to try to re-use buffers and keep the buffer size smaller than the processor cache size.

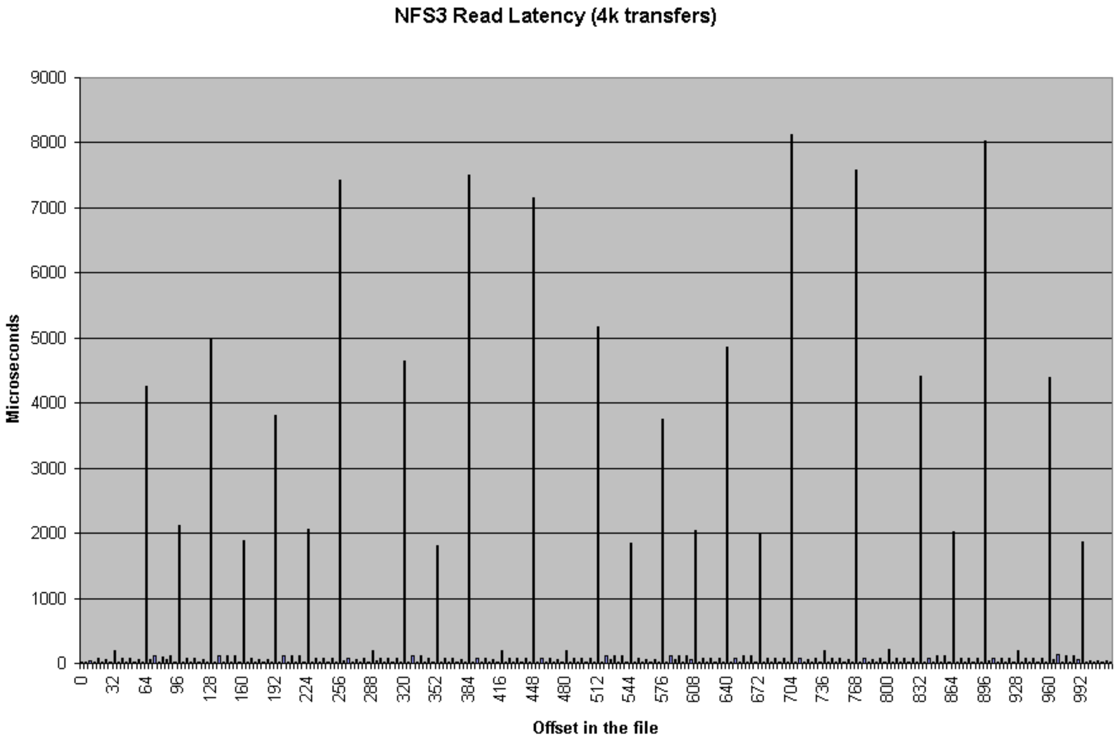


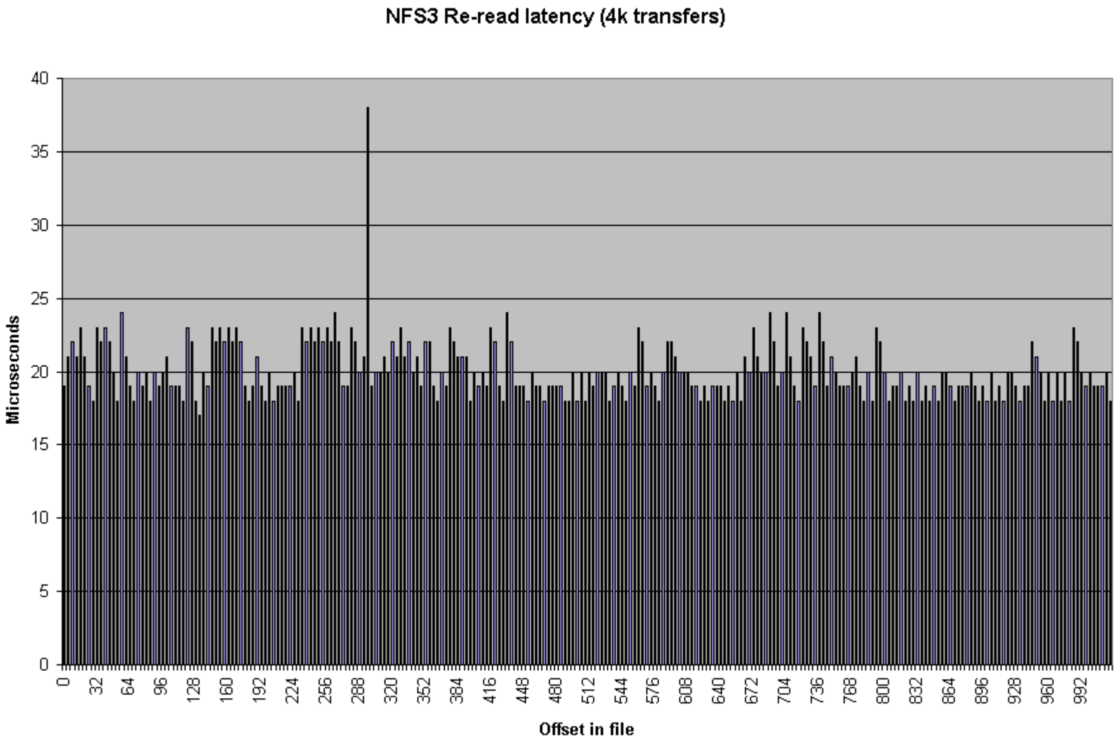
The graph above is an example of a real system with some interesting "optimizations". Here one can see that there are some file sizes and some record sizes that have very bad performance. Notice the performance dip at record sizes of 128Kbytes. (Anomaly #1) There is also a dropoff for file sizes of 8 MB and larger. The dropoff for files greater than 8MB is very interesting since this machine has 16 GB of memory and an 8GB buffer cache. This is a classic example of tuning for a specific application. If the poor system administrator ever installs an application that likes to read or write files in a record size of 128 Kbytes to 1 Mbyte his users will probably take him out back for a conference. If the system would have been characterized before it was purchased it would never have made it into the building.

Another type of graph that can be produced is the Latency graph. When the -Q option is used Iozone will generate four .dat files. Rol.dat, wol.dat, rwol.dat and rrol.dat. These are read offset latency, write offset latency, rewrite offset latency and reread offset latency. These files can be imported into Excel and then graphed.
The latency versus offset information is useful for seeing if there are any particular offsets in a file that have high latencies. These high latencies can be caused by a variety of causes. An example would be if the file size is just a bit bigger than the buffer cache size. The first time the file is written the latency will be low for each transfer. This is because the writes are going into the buffer cache and the application is allowed to continue immediately. The second time the file is written the latencies will be very high. This is due to the fact that the buffer cache is now completely full of dirty data that must be written before the buffer can be reused. The reason that this occurs when the file is bigger than the buffer cache is because the write to the first block on the rewrite case will not find the block in the buffer cache and will be forced to clean a buffer before using it. The cleaning will take time and will cause a longer latency for the write to complete. Another example is when the filesystem is mounted from a remote machine. The latency graphs can help to identify high latencies for files that are being accessed over the network. The following are a few latency graphs for file I/O over an NFS version 3 filesystem.


NFS3 Write latency (4k transfers)
Microseconds
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000
Offset in file
0
32
64
96
128
160
192
224
256
288
320
352
384
416
448
480
512
544
576
608
640
672
704
736
768
800
832
864
896
928
960
992


NFS3 Rewrite latency (4k transfers)
Microseconds
0
100
200
300
400
500
600
700
800
900
1000
Offset in file
0
32
64
96
128
160
192
224
256
288
320
352
384
416
448
480
512
544
576
608
640
672
704
736
768
800
832
864
896
928
960
992


NFS3 Read Latency (4k transfers)
Microseconds
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
32
64
96
128
160
192
224
256
288
320
352
384
416
448
480
512
544
576
608
640
672
704
736
768
800
832
864
896
928
960
992
Offset in the file


NFS3 Re-read latency (4k transfers)
Microseconds
40
35
30
25
20
15
10
5
0
0
32
64
96
128
160
192
224
256
288
320
352
384
416
448
480
512
544
576
608
640
672
704
736
768
800
832
864
896
928
960
992
Offset in file

In the re-read latency graph one can clearly see the client side cache that is in NFS Version 3. The reread latencies are clearly not the latencies that one would get if the reads actually went to the NFS server and back.

Run rules:

If you wish to get accurate results for the entire range of performance for a platform you need to make sure that the maximum file size that will be tested is bigger than the buffer cache. If you don't know how big the buffer cache is, or if it is a dynamic buffer cache then just set the maximum file size to be greater than the total physical memory that is in the platform.
In general you should be able to see three or four plateaus.

File size fits in processor cache.
File size fits in buffer cache
File size is bigger than buffer cache.

You may see another plateau if the platform has a primary and secondary processor caches. If you don't see at least 3 plateaus then you probably have the maximum file size set too small. Iozone will default to a maximum file size of 512 Mbytes. This is generally sufficient but for some very large systems you may need to use the –g option to increase the maximum file size. See the file Run_rules document in the distribution for further information.

Source code availability

Iozone is in public domain and its source is available for free. One might consider using it before your company purchases its next platform.

Additional notes on how to make the graphs

Iozone sends Excel compatible output to standard out. This may be redirected to a file and then processed with Excel. The normal output for Iozone as well as the Excel portion are in the same output stream. So to get the graphs one needs to scroll down to the Excel portion of the file and graph the data in that section. There are several sets of graph data. "Writer report" is one example. When importing the file be sure to tell Excel to import with "delimited" and then click next, then click on the "space delimited" button. To graph the data just highlight the region containing the file size and record size and then click on the graph wizard. The type of graph used is "Surface". When the next dialog box pops up you need to select "Columns". After that the rest should be straight forward.

Contributors: http://www.iozone.org
Original Author: William D. Norcott. wnorcott@us.oracle.com
Features & extensions: Don Capps capps@iozone.org